避けゲー

実習課題　避けゲー

目的（これを体験で学ぶ）

- ゲームエンジニアリングの基礎を知る
- C++ と Visual Studio で実際に動くゲームを作る流れを体験する
- ソースコード → ビルド → 実行ファイル という開発の基本パイプラインを理解する
- プログラミングの基本（変数／順次・分岐・反復の3つの構造）を使えるようになる
- エラーが出たときの「原因を調べて直す」デバッグの流れを身につける
- DxLib（ゲーム用ライブラリ）を使って画面に描画し、ゲームの構成要素を組み立てる

ゲーム構成（この体験で作る「避けゲー」の要素）

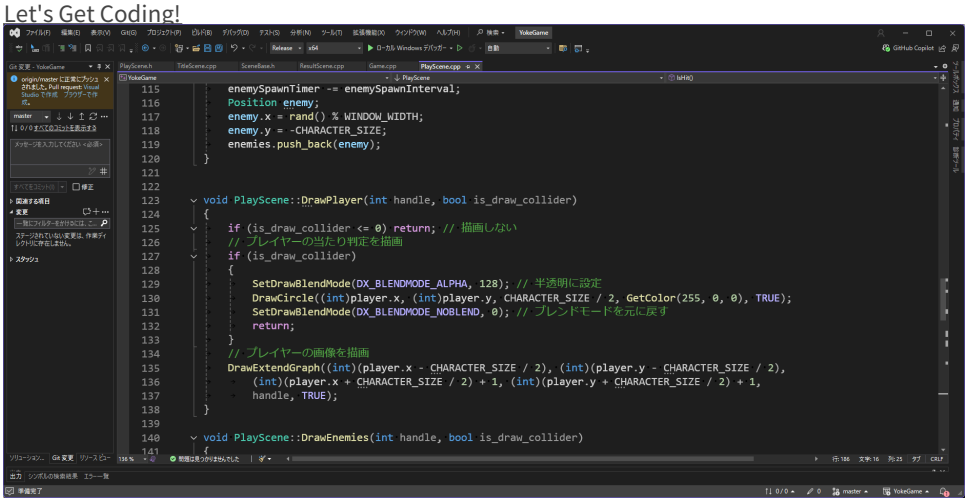
実行してみてみよう

1. タイトル画面（スペースキーでスタート）
2. プレイヤー（左右キーで移動）
3. 敵（上からランダムに落ちてくる）
4. 衝突判定（ぶつかったらゲームオーバー）
5. スコア（何秒生き残ったか）
6. リザルト画面（結果表示）

実習課題（ゲームをうごかしてみよう）

ここから先は、実習ステップ（やってみよう）

Let's Get Coding!

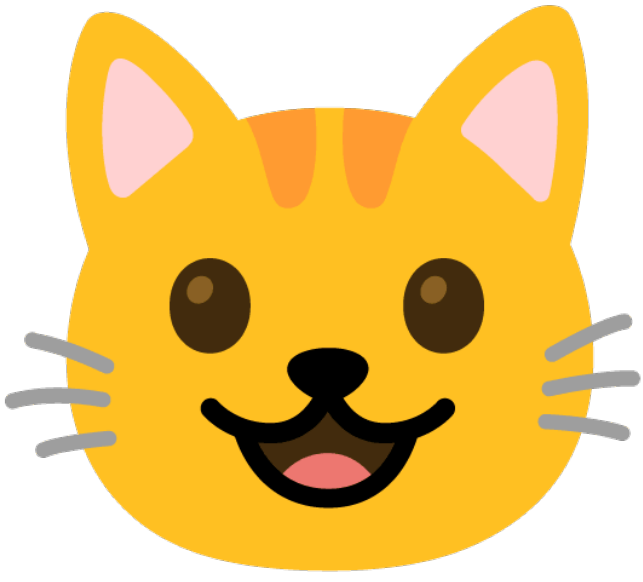
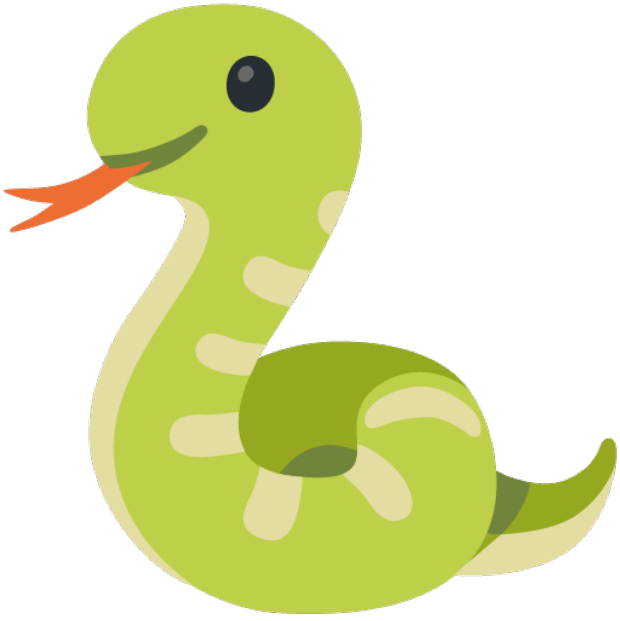


```
        enemySpawnTimer -= enemySpawnInterval;
        Position enemy;
        enemy.x = rand() % WINDOW_WIDTH;
        enemy.y = -CHARACTER_SIZE;
        enemies.push_back(enemy);
    }


void PlayScene::DrawPlayer(int handle, bool is_draw_collider)
{
    if (is_draw_collider <= 0) return; // 描画しない
    // プレイヤーの当たり判定を描画
    if (is_draw_collider)
    {
        SetDrawBlendMode(DX_BLENDMODE_ALPHA, 128); // 半透明に設定
        DrawCircle((int)player.x, (int)player.y, CHARACTER_SIZE / 2, GetColor(255, 0, 0), TRUE);
        SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0); // ブレンドモードを元に戻す
        return;
    }
    // プレイヤーの画像を描画
    DrawExtendGraph((int)(player.x - CHARACTER_SIZE / 2), (int)(player.y - CHARACTER_SIZE / 2),
        (int)(player.x + CHARACTER_SIZE / 2) + 1, (int)(player.y + CHARACTER_SIZE / 2) + 1,
        handle, TRUE);
}

void PlayScene::DrawEnemies(int handle, bool is_draw_collider)
{
```

課題01：画像を読み込んで表示させよ

- 説明: プレイヤーと敵の画像を正しく読み込まないと表示されない。`playerImage` に "image/cat.png"、`enemyImage` に "image/snake.png" を読み込む作業を行う。

読み込み画像

	
cat.png	snake.png

読み込む画像ファイルを指定しよう

✔36~41行目辺り

```
if( playerImage == -1 )
```

```
    playerImage = LoadGraph( "image/XXX.png" );
if( enemyImage == -1 )
    enemyImage = LoadGraph( "image/XXXX.png" );
```

注意点

- 画像ファイルがプロジェクト内に存在することを確認する。パスのスペルミスに注意。

やること

1. `Init()` 内の `LoadGraph` 呼び出しを上記のように書き換える。
2. ビルド & 実行してプレイヤーと敵の画像が表示されるか確認。
3. 表示されなければファイル名やパスをチェック。

確認ポイント

- プレイヤーと敵の画像が画面に表示されているか？
- 画像が表示されない場合、ファイルパスの間違いやファイルの存在を確認したか？

課題02：プレイヤーの移動速度を設定しよう

スピードを表す変数に値を入れよう

✔23行目辺り

C++（他のどの言語でもそうだけど）プログラミングでは、いろいろなものを**変数**として表現します。
プレイヤーの位置は、初期位置と、スピード、経過時間などの**変数**から毎フレーム計算されます。

```
//課題02 プレイヤーの移動速度を設定
playerSpeed = __________f; // 例: 150.0f
```

- 左右キーを押したときに動く量が変わる。値を変えても動きを試してみよう。

> ℹ 用語と基礎説明

- **プログラミングの変数と数学の変数の違い**：数学で習う や は「値がわからないもの（代入される記号）」として出てくるけど、プログラムの変数は「値を入れておける箱」で、中の値を変えたり取り出したりできる。
  例：`playerSpeed` という箱に `150.0f` を入れておいて、使うときにその中身を取り出して移動に使う。
- **座標軸**：画面の左上が (0,0) で、右に行くほど X が増え、下に行くほど Y が増える。

```
(0,0) ┌──────────────────────> X
      |
      |
      v
      Y
```

- **ピクセル**：画面を構成する最小の点。位置やサイズはピクセル単位で考える。
  例：画像が 64x64 ピクセルなら、横幅が64ピクセル、高さが64ピクセル。
- **画像サイズ**：キャラクター画像の幅と高さ。描画位置は中心を基準にしているので、`player.x - CHARACTER_SIZE/2` などで左上を計算している。
- **移動速度（playerSpeed）**：1秒あたり何ピクセル動くかを表す。`player.x += playerSpeed * deltaTime;` で実際の移動距離になる。
  例：`playerSpeed = 150.0f;` なら 1秒で 150 ピクセル移動する（0.5秒なら 75 ピクセル）。

課題03：敵の落下速度を設定しよう

プレイヤーは動き始めたけど、今度は敵が落ちてきません。

敵のスピードも、（わざとらしく？）0になってるので、スピードを設定してあげてください。

敵の落下速度を設定しよう

✔26行目辺り？

```
//課題03 敵の落下速度を設定
enemySpeed = _____f; // 例: 150.0f
```

- 敵がどれくらい速く落ちてくるかを調整する。難易度に関係する。
- 今の画面のサイズは横×縦＝960x640、640ピクセルを2秒で落ちるにはスピードは？みたいに決める

課題04：敵の生成（出現）タイマーを増やそう

敵がまだ出現しない。。。
敵は、一定時間ごとに設定されたタイマーによって、定期的に現れる仕組みになっている。
現状では、タイマーが動いていないのでタイマーを動かして敵を出現させよう！

ここでは、ゲームの進行管理で重要な**フレーム**について学ぶよ！

✔59行目辺り？

```
//課題04 敵の生成タイマーを加算
enemySpawnTimer = enemySpawnTimer + deltaTime; // ここを書いて敵が定期的に出るようにする
```

- `enemySpawnInterval` を超えたら `CreateEnemy()` が呼ばれる。

> ℹ 仕組みの補足（フレームごとの動きとタイマーの関係）

- ゲームは**「フレーム」**という単位で動いている。1フレームごとに画面を更新する。普通は1秒に60回くらい（60FPS）呼ばれる。
- 毎フレーム、次の順番で処理が行われる：
  1. `Update()`（状態を変える：敵を落とす、タイマーを増やす、入力を読む）
  2. `Draw()`（画面に今の状態を描く）
- `deltaTime` は前のフレームからの時間（たとえば 1/60 秒＝約0.0167）。これを `enemySpawnTimer` に足して「時間を数える」。
- `enemySpawnTimer` が設定された間隔（`enemySpawnInterval`）を超えたら、その時点で新しい敵が出現する。つまり「何秒たったら出すか」をフレームをまたいで数えている。
- 例：`enemySpawnInterval = 0.5f` なら、0.5秒ごとに敵が出る。60FPS なら約30フレームごとに `CreateEnemy()` が呼ばれる。

> ⚠ 式 `enemySpawnTimer = enemySpawnTimer + deltaTime` について（数学とプログラムの違い）

- 数学の式だと左辺と右辺に同じ記号が出てくると変に見えるが、プログラムでは「**今の値に新しい分を足して更新する**」操作として普通に使う。
  これは「箱の中身を取り出して、そこに時間分を足して、また箱に戻す」処理と考えられる。
  例：今のタイマーが0.3秒で、`deltaTime` が0.0167なら、次のフレームでは 0.3167 秒になる。

課題05：スコアの更新を有効にしよう

フレーム間時間(deltaTime)を使ってスコアの変数を変化させよう

✔93行目辺り

```
//課題05 スコアの値を更新
score = score + ?????; // 生き残った時間がスコアになる
```

仕組みの補足（敵の出現タイマーと同じ考え方）

- スコアも `enemySpawnTimer` と同じで、毎フレーム `deltaTime` を足して時間を数えている。
- `score += deltaTime;` は「生き残った時間」がそのまま点数になる仕組み。60FPSなら1フレームごとに約0.0167ずつ増える。

画面表示の豆知識

- このスコア表示や文字の描画は `DxLib` が簡単にしてくれているから少ないコードで書ける。
  もし `DirectX` 単体で文字を表示しようとすると、フォントを読み込んでテクスチャを作り、描画用の頂点バッファやシェーダを用意する必要があり、かなりコードが長くなる。
  `DxLib` はそういう面倒な下準備を隠してくれていて、`DrawFormatStringToHandle` などを呼ぶだけで表示できる。

課題06：当たり判定をつけよう

当たり判定が働いたら敵を消す／終了にする（条件分岐、関数呼び出し）

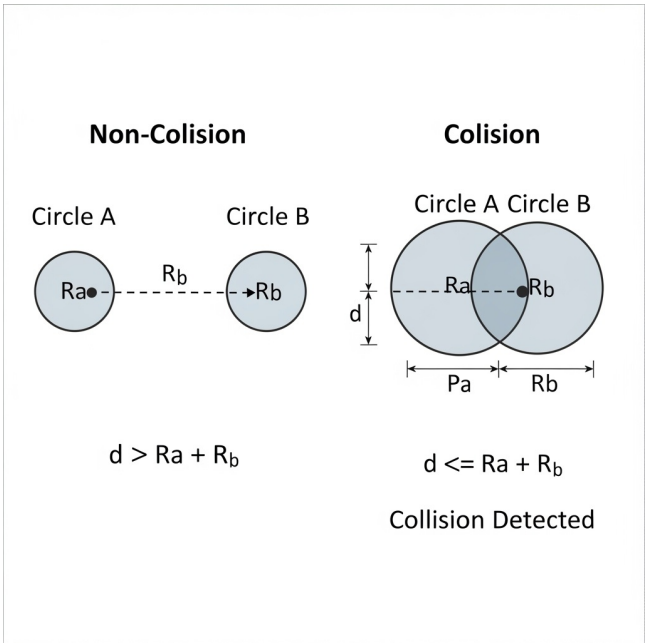
✔86行目辺り

当たり判定の処理を呼び出してみよう。当たり判定の処理は、別のところに `IsHit()` として記述されています。
それを必要なところで呼び出すと、すべての敵と、プレイヤーがぶつかっているかどうかを判定し、その結果を真偽値（true, false）で、知らせてくれます。

```
//課題06 当たり判定
if( 0 )  //ここで当たり判定の処理を呼び出してみよう
{
   ClearEnemies(); // 敵を消してみる（実装済み関数を使う）
   // 課題07 リザルト画面へ移行（下につなげる）
}
```

説明：ついに当たり判定です

ゲームにおける当たり判定とは、**画面上のもの同士が「ぶつかったかどうか」**を調べて、**それに応じた反応**（ダメージを受ける／ゲームオーバーになる／アイテムを取るなど）**を起こす**仕組みです。
　このプログラムでは、キャラクター（プレイヤー）と敵のぶつかり判定に**「円と円の距離」**を使っています。
プレイヤーと敵の中心の距離を計算して、それが一定以内（キャラクターのサイズの2乗）ならぶつかったと判断します。これは簡単で速い衝突チェックの方法です。



条件分岐(if文)と真偽値

- 当たり判定の結果は「ぶつかっているか」「ぶつかっていないか」の **真（true）／偽（false）** で表される値（真偽値）として返される。`IsHit()` はこれを返す関数で、ぶつかっていれば true を、そうでなければ false を返す。
- `if( IsHit() ) { ... }` のように書くと、「ぶつかったときだけ中の処理をする」ことができる。これが **条件分岐** で、プログラミングの3つの基本構造（順次・分岐・反復）のひとつで非常に重要。処理を状況に応じて切り替えるための仕組み。
- ここでは「当たっていたら敵を消す／ゲームオーバーにする」といった反応を `if` を使って実現している。

サブルーチン（関数）としての `IsHit()` の役割

- `IsHit()` という関数が別に作られていて、毎フレームの中で呼ばれています。
  ゲームの処理は1フレームごとに `Update()` → `Draw()` の順に動いていて、その `Update()` の中で状態チェック（衝突の有無など）を行うために `IsHit()` を呼び出します。
- このように「特定の処理をまとめて名前をつけ、必要なときに呼び出す仕組み」を **サブルーチン**（関数、ルーチン）と呼びます。
  サブルーチンを使うと、同じ処理を何度も書かずにすみ、プログラムが整理されて読みやすく、直しやすくなります。

課題07：リザルト画面へ移行する処理を追加しよう

✔ ８９行目辺り

```
//課題06 当たり判定
 if( 0 )
 {
  ClearEnemies();
  //課題07 リザルト画面へ移行
  //->ここに移行処理を追加
 }
```

- 衝突後にリザルト用のシーンに変えるコードを追加。
- **補足**：`GoToResultScene()` という関数が用意されている

> **画面遷移と Scene（状態遷移）の考え方**

- 多くのゲームは「Scene（シーン）」という単位で画面ごとの役割を分けて管理している。たとえばタイトル画面、プレイ画面、リザルト画面がそれぞれ別の Scene。
- ある条件が満たされると Scene を切り替える。これを **状態遷移** と呼ぶ。たとえば当たり判定で衝突したら「プレイ中」から「リザルト」へ移る、といった具合。
- `GoToResultScene()` や `changeScene(...)` は状態を変える関数で、現在の Scene の状態を終了して新しい Scene に移る（遷移する）処理をまとめている。
- 状態遷移は画面遷移だけでなく、**キャラクターの状態管理（立っている・走っている・ジャンプ中など）やアニメーションの切り替え**、敵のAIの振る舞いの切り替えなど、ゲームのあらゆる部分で使われている。
- 状態遷移を整理しておくと、「何が起きたら次に何を表示するか／どう振る舞うか」が明確になり、ゲーム全体の流れや作りやすさが向上する。

おまけ

このゲームの「動く」仕組みのミニ解説

- プログラミングの３つの基本構造
  1. **順次**：コードは上から下へ動く。初期化 → 毎フレームの処理（Update） → 描画（Draw）と流れる。
  2. **分岐**：`if` で条件によって違うことをする（例：左右キーで動くかどうか）。
  3. **反復**：`for` で敵を全部落とす、全部描く。
- **変数**：`player.x` や `score` は変化する値を保存する箱。
- **deltaTime**：前のフレームからの時間。これを掛けるとどんなPCでも速さが同じになる。
- **当たり判定**：プレイヤーと敵の距離の2乗を比べてぶつかったか判定している。

主な変数の意味（実習で触るもの）

- `player.x`, `player.y`：プレイヤーの位置。画面上のピクセル座標（左上が0,0）。
- `playerSpeed`：プレイヤーが1秒あたり何ピクセル動くか。左右キー入力時に使われる。
- `enemySpeed`：敵が1秒あたり何ピクセル落ちるか。落下の速さ。
- `enemySpawnInterval`：何秒ごとに新しい敵を出すかの間隔。小さいほど敵がたくさん出る。
- `enemySpawnTimer`：出現までの経過時間をためておくタイマー。これが `enemySpawnInterval` を超えると敵が出る。
- `score`：生き残った時間を蓄積した値。時間経過で増える（＝ゲームの得点）。
- `enemies`：敵の位置のリスト。`for` ループで全部更新・描画・判定する。
- `CHARACTER_SIZE`：プレイヤー・敵のサイズ（直径）。当たり判定や描画位置の計算に使う。

フレームレート依存と「距離 = 速さ × 時間」の重要性

- もし `player.x += playerSpeed;` のように **deltaTime を掛けずに**移動処理を書いた場合、1フレームあたり同じ量だけ動くので、フレームレートが高いと速く、低いと遅くなる。
  例：1秒間に 60 フレームだと 60 × 150 = 9000 ピクセル動くが、30 フレームだと 30 × 150 = 4500 ピクセルしか動かない。
  動きがマシン依存になってしまう。
- これを防ぐには「距離 = 速さ × 時間」の考え方で、`player.x += playerSpeed * deltaTime;` のように、前のフレームからの時間（deltaTime）を使って移動量を調整する。
  こうすると、どんなフレームレートでも1秒あたり同じ距離だけ動く。
- `deltaTime` を使うのはゲームでもっとも基本的なテクニックの一つ。

なぜこれを使わないと困るのか（詳しい影響）

1. **動きがマシンによって違う**：高性能なPCではフレームレートが高くなりすぎてキャラクターが速くなり、古いPCでは遅くなる。プレイ体験がバラバラになり、公平なゲームにならない。
2. **入力の反応や操作感が不安定になる**：フレームが速いとキーを押した瞬間の移動量が大きく変わるため、プレイヤーが思った通りに操作できない。
3. **物理的な計算が壊れる**：重力や速度の積み重ねなど、時間に依存するシミュレーションはフレームごとに変わると累積誤差やバウンスの違いが発生し、動きがぎこちなくなる。
4. **再現性がなくなる**：同じ操作をしてもフレームレートによって結果が変わるので、バグの再現や調整が難しくなる。
5. **フレーム落ちで急に遅くなる現象（スタッタリング）**：一時的にフレームレートが下がると移動距離も一気に減る／飛び跳ねたように見えることがある。

✔追加の対策例

- **固定タイムステップ**を使って、物理計算だけ一定の時間間隔で何度も更新し、その間の描画は自由に行う方法。時間の積み残しを管理して見た目と計算を分けることで安定した動きにできる。
- **補間**を使って描画と物理のズレをなめらかに見せる工夫をする。

発展課題

ゲームの中で気になるところない？

- プレイヤーが画面の外に出るまで移動可能
  - どうやって、移動を制限するかな？
- 時間があったら試してみよう
  - 敵の見た目の変更
  - 敵の出現間隔の変更
  - 敵の落下位置をプレイヤーキャラクター付近に変更
  - 敵の落下スピードを徐々に上げる

Revision #14
★Created 30 July 2025 16:33:41 by youe2
Updated 12 June 2026 09:23:17 by youe2